High Performance Matrix-Fee Method for Large-Scale Finite Element Analysis on Graphics Processing Units

by

Petros Apostolou

Undergraduate Diploma in Mechanical Engineering

National Technical University of Athens (NTUA), 2015

Submitted to the Graduate Faculty of

Swanson School of Engineering

in partial fulfillment

of the requirements for the degree of

Master of Science

University of Pittsburgh

2020

UNIVERSITY OF PITTSBURGH

SWANSON SCHOOL OF ENGINEERING

This thesis was presented

by

Petros Apostolou

It was defended on

April 2, 2020

and approved by

Albert To, Ph.D, Professor, Department of Mechanical Engineering and Materials Science

Senocak Inanc, Ph.D, Associate Professor, Department of Mechanical Engineering and Materials Science

Patrick Smolinski, Ph.D, Associate Professor, Department of Mechanical Engineering and Materials Science

Thesis Advisor: Albert To, Ph.D, Professor, Department of Mechanical Engineering and Materials Science

High Performance Matrix-Fee Method for Large-Scale Finite Element Analysis on Graphics Processing Units

Petros Apostolou, M.S.

University of Pittsburgh, 2020

This thesis presents a high performance computing (HPC) implementation on graphics processing units (GPU) for large-scale numerical simulations. In particular, the research focuses on the development of an efficient matrix-free conjugate gradient solver for the acceleration and scalability of the steady-state heat transfer finite element analysis (FEA) on a three-dimensional uniform structured hexahedral mesh using a voxel-based technique. One of the greatest challenges in large-scale FEA is the availability of computer memory for solving the linear system of equations. Like in large-scale heat transfer simulations, where the size of the system matrix assembly becomes very large, the FEA solver requires huge amounts of computational time and memory that very often exceed the actual memory limits of the available hardware resources. To overcome this problem, in this work a matrix-free conjugate gradient (MFCG) method is implemented to finite element computations which avoids the global matrix assembly. The main difference of the MFCG to the classical conjugate gradient (CG) solver lies on the implementation of the matrix-vector product operation. Matrix-vector operation found to be the most expensive process consuming more than 80% out of the total computations for the numerical solution and thus a matrix-free matrix-vector (MFMV) approach becomes beneficial for saving memory and computational time throughout the execution of the FEA. In summary, the MFMV algorithm consists of three nested loops: (a) a loop over the mesh elements of the domain, (b) a loop on the element nodal values to perform the element matrix-vector operations and (c) the summation and transformation of the nodal values into their correct positions in the global index. A performance analysis on a TITAN V GPU between the parallel MFCG and NVIDIA Amgx matrix-assembly solver shows that the MFCG solver is 2.12 $\times$ times faster consuming 7.76 $\times$ times less memory. The parallel MFCG GPU solver achieves a maximum of 81.81 $\times$ speed-up allowing for FEA of the steady-state heat-conduction for mesh sizes up to $550^3 = 166.375M$ elements.

Table of Contents

List of Tables

List of Figures

Preface

I want to acknowledge all the factors of the University of Pittsburgh and especially the Swanson School of Engineering for the completion of my Master of Science degree in the Department of Mechanical Engineering and Materials Science. Many thanks to the director of the program Dr Inanc Senocak and the previous department's director Dr Qing-Ming Wang for their support throughout the program. I would also like to thank the committee of my Master thesis defence consisting of the Drs Senocak Inanc, Patrick Smolinski and Albert To.

Special thanks are going to my advisor Dr Albert To for the opportunity that he offered me to join the Lab of Modeling & Optimization Simulation Tools for Additive Manufacturing (MOST-AM). Here, I would like to thank all the students and researchers of the MOST-AM laboratory and especially the post-doctoral researcher Dr Florian Dugast for his continuous advise and support during the research and dissertation of my master research project.

I am also very happy to give a very special thank to my friend and PhD candidate Aidyn Aitzan for his overall help, support and advise throughout my program and research. Last, i want to thank very much the family of Dr Thanos Tzounopoulos director of the Pittsburgh Hearing Research Center, Endowed Professor and Vice Chair of Research of the department of Otolaryngology for all the support and encouragment they offered to my graduate studies.

"One i do know...that i do know nothing" (Socrates 470 - 399 *B.C)*

1.0 Introduction

1.1 Thesis Outline

The context of this thesis is formed to three main parts. The first part describes the general finite element formulation of the steady-state heat conduction problem. In the second part an efficient matrix-free conjugate gradient solver for the finite element analysis of the steady-state heat transfer conduction is presented. At the last part, this study demonstrates the performance of an efficient matrix-free method in terms of computational time and memory usage of both central processing unit (CPU) and graphics processing units (GPU) implementations of the matrix-free conjugate gradient method (MFCG) for computing the solution of the linear system $KT = F$. Here K is the conductivity matrix, T the unknown field of the temperature, and F the heat source. The performance of the matrix-free method is compared with the traditional global conductivity matrix assembly for different mesh sizes.

1.2 Background on Finite Element Computations

When solving partial differential equations using the finite element method, the standard procedure consists of two distinct steps, a) an assembly step, where a system matrix and right hand side vector are created to form a linear system KT=F, and b) a solution step, where the linear system is solved. In most problems, computational time for solving the linear system is dominant, and therefore, parallelization of the solution step becomes the main factor for achieving the acceleration of the solution of the linear system.

A very related problem shows up when using the finite element method with a matrix-free approach, where the assembly phase becomes uneccessary and is replaced by the element sparse matrix-vector product inside the algorithm of the linear solver. Utilizing the matrix-free algorithm for the solver, the explicit global system matrix assembly can be avoided. Since the matrix-vector operation is performed with a higher frequency throughout the simulation,

efficient implementation of this operation is crucial. This matrix-free approach is motivated by the limited memory available for the system matrix assembly, but also by the computation of the numerical solution in large size domains (e.g 100M mesh elements). In the matrix-free methods, Ref. [1] the matrix-vector product operations are computed as local summations of contributions from all elements in the finite element mesh, where the contributions from a single element correspond to the degrees-of-freedom residing within that element. To this end, the matrix-vector multiplication is performed by computing the summation of the contributed matrix-vector operations of the nodes that are shared between neighboring mesh elements. In steady-state heat conduction FEA using the matrix-free method, the small size of the local conductivity matrix, $Ke[8\times8]$ which is considered for the matrix-vector operation becomes the main advantage of the matrix-free methods over the traditional global system-assembly approaches where the much larger size of the global conductivity matrix, $K[N\times N]$ takes place into the algorithm for calculating the solution. Indeed, matrix-free methods require more floating point operations than the global matrix-assembly ones because of the extra operations are needed for the accumulative summations of the local nodal values. The main gain of the matrix-free method comes from the fact that each operation is computed very fast characterized by a compute-bound behavior (performance is limited by the rate at which the processor can perform arithmetic operations) rather than a bandwidth-bound (memory bandwidth is the limiting factor of the computations) [73]. The later one is typically the case in FEA solvers using the global system matrix-assembly step. This compute-bound behavior of the matrix-free method leads to higher computational intensity and efficiency. Especially for applications within a limited memory of a GPU device (e.g 12GB RAM), matrix-free method can also achieve an improved scaling of the numerical solution as the size of the FEA problem increases.

1.3 State of the Art of Matrix-Free Methods

The finite element method (FEM) is widely used for numerical simulation of partial differential equations (PDEs) in engineering applications. Usually the finite element analysis

(FEA) is computed using algebraic multi-grid (AMG) [9] or preconditioned conjugate gradient iterative (PCG) [20] solvers based on the global system matrix assembly of the linear system. In large-scale FEA the global system matrix assembly requires huge amounts of memory and the calculation of the numerical solution becomes a very expensive process. For example, in the heat transfer FEA of a layer-by-layer process simulation for the Laser Powder Bed Fusion (L-PBF), Ref. [56] the global matrix assembly has to be updated for each layer during the solution step and for realistic mesh sizes this becomes a very time-consuming process. To reduce the computational time and memory usage of the computation of the finite element solution researchers have been developing matrix-free methods that show an improved scalability over the size of the computational mesh. Through their regular structure grid-based models, matrix-free algorithms are able to compute the solution with techniques which store one basic element stiffness matrix and calculate the matrix-vector product locally on either the element [28] or the node [23] level. Replacing global-assembly with small matrix-vector products, the memory restriction is overcomed through the drastic reduction of the memory demand and thus, larger-sized models can be solved on modern desktop computers. Considering iterative-based solvers like the conjugate gradient, it has been observed that the most time-consuming step in each iteration of the algorithm is the matrix-vector product. This product in matrix-free methods is computed without the need of assembling and saving the global system matrix. The idea builds on the observation that, to solve the linear system $KT = F$ using an iterative method, the system matrix K is never needed explicitly, only its effect as an operator $K[\,\cdot\,]$ on a vector T. This, means that a recipe KT can be used to solve the system $KT = F$ without having access to an explicit K.

In most widely used commercial FEA software the matrix-free solution algorithm is not included and the advantages of regular voxel-based models cannot be applied. Matrix-free solution techniques were first used by [1] , and nowadays, matrix-free methods are applied for many large scale problems e.g. for offshore structures [2], for seismic wave modeling [3], for a time-transient proliferation of cellular tissue and for simulation of laser processing of particulate-functionalized materials [4]. Moreover, Melenk et al. [46] investigate efficient techniques for assembly of the spectral element stiffness matrix, based on a tensor-product approach. Cantwell et al. [47] compare matrix-free techniques based on a local matrix and

tensor-product evaluation, with the standard sparse-matrix approach, and find that the optimal approach depends on both the problem setup and the computer system. Kronbichler and Kormann [9] describe a general framework implementing tensor-based operator application parallelized using a combination of MPI for inter-node communication, multicore threading using Intel Threading Building Blocks, and SIMD vector intrinsics. The framework has been included in the open-source finite-element framework deal.II Ref. [57]. A multigrid solver for Poisson equations with a matrix-free implementation applied to FEA and variants of the discontinuous Galerkin method is up to order of magnitude faster compared to matrix-based implementation due to vastly better performance of matrix-free operation compared to sparse matrix [9]. In Ref. [10] also a comparison between matrix-free methods is presented and it is found that for hexahedron the an element-by-element proccedure is particularly suited and for unstructured grids the edge-by-edge method is superior to element-by-element solutions for viscous-plastic flows in Ref. [11] as well as for solid mechanics in Ref. [12].

The matrix-free method can be also found in fluid mechanics applications where a matrix-free procedure is applied to the implicit finite volume lattice Boltzmann method [5], to incompressible Navier-Stokes equations [6], to heterogeneous Stokes flow [7] and to Galerkin formulations of compressible flows [8].

1.4 Introduction of the Matrix-Free Method

The matrix-free method algorithm is developed using an element-by-element approach to the matrix-vector multiplication operations and is integrated into the conjugate gradient iterative solver. In chapter 3, the serial version of the matrix-free conjugate gradient solver is introduced for the numerical simulation of the steady-state heat conduction. The steady-state heat conduction simulation is computed using structured and uniform hexahedral meshes but the same approach can be applied to non-uniform grids assuming the connectivity index is given. Chapter 4 presents the parallel implementation of the matrix-free conjugate gradient on graphical processing units and discusses the main key points of the computational techniques for GPU parallelization. In chapter 5, the numerical experiments

executed on both a serial CPU and a parallel GPU version are showing a better performance of the matrix-free method over the global matrix assembly both in computational time and memory usage. The scalability of the matrix-free algorithm is tested on larger mesh sizes up to 550^3 mesh elements. The numerical results confirm the efficiency of the matrix-free method for realistic sizes close enough to industrial-scale simulations. Last chapter outlines the basic findings of this study and discusses future applications of the matrix-free method in the field of large-scale FEA simulations.

1.5 Graphics Processing Units

In recent years, programming of graphics processing units (GPUs) for general computations has become very popular. Driven by the insatiable demand from the gaming market for ever more detailed graphics [58], the GPUs have evolved into highly parallel streaming processors capable of performing hundreds of billions of floating point operations per second. The design of GPUs is streamlined to the nature of their workload. Computer graphics essentially consists of processing a very large number of independent polygon vertices and screen pixels. Because of the very large number of tasks, there is no problem with executing each individual task slow as long as the overall throughput is high. Therefore, most of the transistors of a GPU can be used for performing computations. This is in contrast to CPUs, which are expected to perform large indivisible tasks in serial, or a few moderately sized tasks in parallel, possibly with complicated inter-dependencies. To optimize for this workload, i.e. making a single task finish as quickly as possible, a considerable amount of the hardware of a CPU in fact most of it is dedicated to non-computation tasks such as cache, branch prediction and coherency. Also, to get the necessary data for all the individual work items, the memory system of graphics cards tend to be focused on high bandwidth, whereas the caching system of a CPU aims at achieving low latency. Finally, as computer graphics in many cases can tolerate a fairly low numerical precision, the GPU architecture has been optimized for single-precision operations. This means that while CPUs can typically perform operations in single precision twice as fast as in double precision, this factor is of the order of

3-8 for GPUs. As a consequence of the higher computing power per transistor, GPUs achieve a higher efficiency,both economically (i.e. GFLOPS/$) and power-wise (i.e. GFLOPS/W), although the most recent multicore CPUs are improving in this respect to GFLOPS.

Scientific applications, such as e.g. stencil operations or matrix-matrix multiplications, are usually comprised of many small and similar tasks with a high computational intensity. Because of this, the fit for the throughputoptimized high-bandwidth GPU hardware has in many cases been great. However, several limitations of the graphics-tailored GPU architecture limit how well applications can take advantage of the available performance potential of GPUs. For instance, few applications possess the amount of parallelism needed to saturate the massively parallel GPUs. In addition, for most scientific applications, double precision is necessary to obtain meaningful results, which, as mentioned, has a performance penalty over single precision. Furthermore, while dependencies and synchronization are unavoidable parts of most algorithms, these are often very difficult or even impossible to resolve on GPUs. Thus, in order to fully utilize GPUs, it is often necessary to make substantial changes to existing algorithms, or even invent new ones, which take these limitations into account. Another issue is that data has to be moved to the graphics memory before it can be accessed by the GPU, which is presently done by transferring the data over the relatively slow PCI express bus (PCIe). To avoid this bottleneck, data is preferably kept at the GPU for the entire computation. Another approach is to hide the latency by overlapping computation and data transfer.

The history of general-purpose graphics-processing unit (GPGPU) computations started around 2000 when dedicated graphics cards were becoming mainstream. In the beginning, the general-purpose computations had to be shoehorned into the graphics programming pipeline by storing the dataas textures and putting the computations in the programmable vertex and pixel shaders. Examples of early successful general-purpose computations on graphics hardware are matrix-matrix multiplication [15], a solution of the compressible Navier-Stokes equations [16], and a multigrid solver [17]. A summary of early work in GPGPU can be found in the survey paper by Owens et al. [18].

However, the many restrictions and the fact that a programming model for graphics had to be exploited made it a daunting task to do general purpose computations with the graphics

pipeline. In response to this, at the end of 2006, Nvidia released CUDA, Compute Unified Device Architecture, which simplified the programming and led to a dramatic increase in interest for GPGPU.

The CUDA platform provides a unified model of the underlying hardware together with a C-based programming environment. The CUDA GPU, or device, comprises a number of Streaming Multiprocessors (SMs) which in turn are multi core processors capable of executing a large number of threads concurrently. The threads of the application are then grouped into blocks of threads and each block is executed on a single SM, independently of the other blocks. Within a thread block or an SM, there is a piece of shared memory. Furthermore, it is possible to have synchronization between the threads of a single block, but it is not possible to synchronize threads across blocks, except for a global barrier. There is also a small cache shared between the threads of a block.

An important feature of CUDA, and arguably the most crucial aspect to attain good utilization of the hardware, is the memory model. Because transfers from the main device memory are only made in chunks of a certain size, and due to poor caching capabilities, it is important to use all the data of the chunks which are fetched. When the threads within a block access a contiguous piece of memory simultaneously, such a coalesced memory access is achieved. For further details on the CUDA platform, see the CUDA C Programming Guide [19]. Examples of fields where CUDA has been successfully utilized include molecular dynamics simulations [20], fluid dynamics [21], wave propagation [22], sequence alignment [23] and, Monte Carlo simulations of ferromagnetic lattices [24]. In response to CUDA and the popularity of GPU programming, OpenCL was launched by the consortium Khronos Group in 2008 [25]. In many respects, such as the hardware model and the programming language, OpenCL and CUDA are very similar. However, in contrast to CUDA, which is proprietary and restricted to Nvidia GPUs, OpenCL is an open standard, and OpenCL code can be run on all hardware with an OpenCL implementation, today including Nvidia and AMD GPUs, and even Intel and AMD CPUs. While the same OpenCL code is portable across a wide range of platforms, it is usually necessary to hand tune the code to achieve the best performance. In addition, CUDA, being made by Nvidia specifically for their GPUs, is still able to outperform OpenCL in comparisons and when optimal performance

is desirable, CUDA is still the natural choice [26], [27]. Critique has been raised as to the long-term viability of techniques and codes developed for GPUs in general and CUDA in particular, since these are very specific concepts which might have a fairly limited life time. However, an important point is that GPUs are part of a larger movement that of heterogeneity and increasing use of specialized hardware and accelerators. Recently, all the major processor vendors have started offering dedicated accelerators for computations, which, in addition to the Tesla GPUs of Nvidia, include Intels Xeon Phi co-processor and the very recently announced FirePro cards by AMD (see Table 3.4). Moreover, there are application programming interfaces (API) like OpenMP Ref. [59] and OpenACC for GPU computing Ref. [60] that offer a portable user control of parallelism. For example, OpenMP can be efficient for intra-node (single node) computations or it can by combined with MPI for inter-node (multi-node) communications forming hybrid parallel schemes running on computer clusters. As in OpenMP 4 or newer, using OpenACC one can annotate C, C++ and Fortran source code to identify the areas that should be accelerated using compiler directives and additional functions. This becomes convenient especially in codes with large number of lines where both OpenMP and OpenACC allow for parallelization and acceleration with minor additions to the code but also flexible in terms of specific compiler dependencies.

2.0 Finite Element Method in Steady-State Heat Conduction

The finite element method (FEM) is a popular numerical method for engineering problems with complicated geometries. It finds applications in thermo-structural mechanics, fluid mechanics, and electromagnetics. Rather than solving the strong form of a partial differential equation, the finite element method is searching for solutions to the variational, or weak, formulation.

In this thesis the three-dimension (3D) finite element formulation is introduced for the steady-state heat conduction equation. The differential form of Fourier's law of thermal conduction is

$$\frac{\partial}{\partial x}\left(\kappa\frac{\partial T}{\partial x}\right)+\frac{\partial}{\partial y}\left(\kappa\frac{\partial T}{\partial y}\right)+\frac{\partial}{\partial z}\left(\kappa\frac{\partial T}{\partial z}\right)+Q(x,y,z)=0, \tag{2.1}$$

where T is the temperature in (x, y, z) spatial direction, κ the thermal conductivity and Q the heat source. By assuming isotropic thermal conductivity $\kappa = 1$ on all directions, equation (2.1) can be written in the following form

$$\frac{\partial^2 T}{\partial x^2}+\frac{\partial^2 T}{\partial y^2}+\frac{\partial^2 T}{\partial z^2}+Q(x,y,z)=0. \tag{2.2}$$

The numerical integration of equation (2.2) in a finite domain $V \in \mathbb{R}^3$ is implemented using the Galerkin method in each element with the corresponding shape function N as

$$\int_V N^{\intercal}\left(\frac{\partial^2 T}{\partial x^2}+\frac{\partial^2 T}{\partial y^2}+\frac{\partial^2 T}{\partial z^2}+Q(x,y,z)\right)dV=0, \tag{2.3}$$

where N are the shape functions and for a tri-linear 8-node hexahedral finite element are given by the following relationship.

$$N_i=\frac{1}{8}(1\pm\xi)(1\pm n)(1\pm\zeta). \tag{2.4}$$

Figure (2.1) shows the mapping of the coordinates for the hexahedral element.

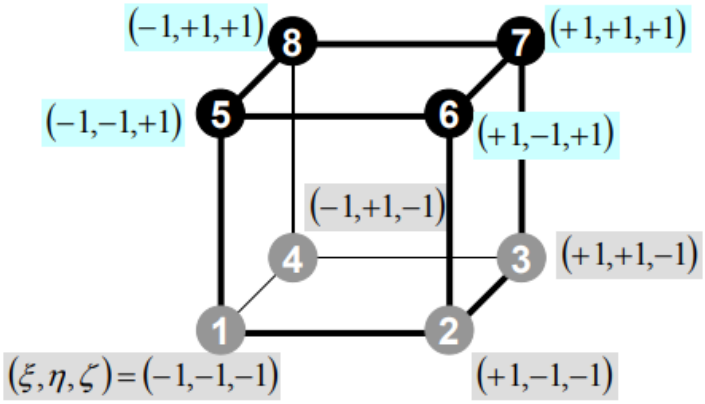


Figure 2.1: Mapping of natural to local elements for the hexahedral finite element.

Considering the terms $T_e = NT$ and $\frac{\partial T_e(x,y,z)}{\partial(x,y,z)} = \frac{\partial N(x,y,x)}{\partial(x,y,z)}T$ with T_e the distribution of the temperature inside the finite element and T the temperature at each one of its nodes, the weak form is obtained as

$$\int_V \left((\frac{\partial N}{\partial x})^\intercal (\frac{\partial N}{\partial x}) + (\frac{\partial N}{\partial y})^\intercal (\frac{\partial N}{\partial y}) + (\frac{\partial N}{\partial z})^\intercal (\frac{\partial N}{\partial z}) \right) dV \int_V T dV = \int_V Q dV \tag{2.5}$$

The integration of the weak form leads to the formation of the finite element system for the steady-state heat conduction on a hexahedral element

$$K_e T_e = F_e, \tag{2.6}$$

where the element conductivity matrix is an 8×8 matrix $K_e = B^\intercal B$ with B the matrix of the derivatives of the shape functions

$$B = \begin{bmatrix} \frac{\partial N_1}{\partial x} & \frac{\partial N_2}{\partial x} & \cdots & \frac{\partial N_8}{\partial x} \\ \frac{\partial N_1}{\partial y} & \frac{\partial N_2}{\partial y} & \cdots & \frac{\partial N_8}{\partial y} \\ \frac{\partial N_1}{\partial z} & \frac{\partial N_2}{\partial z} & \cdots & \frac{\partial N_8}{\partial z} \end{bmatrix}, \tag{2.7}$$

2.1 The Global System Matrix-Assembly Method

The aim of the system matrix assembly is to form the global equation system as a superposition of element equations of the form of equation (2.6).

From equation (2.5) and with the introduction of the following vectors for n grid points, the global conductivity matrix is assembled:

$$\begin{aligned} T[n] &= T_1, T_2, \dots T_n \\ F[n] &= F_1, F_2, \dots F_n \end{aligned} \tag{2.8}$$

$$K[n][n] = \begin{bmatrix} K_{11} & K_{12} & \dots & K_{1n} \\ K_{21} & K_{22} & \ddots & K_{2n} \\ \vdots & \ddots & \ddots & \vdots \\ K_{n1} & K_{n2} & \dots & K_{nn} \end{bmatrix} \tag{2.9}$$

and the global system of equation is constructed

$$KT = F, \tag{2.10}$$

with K the global conductivity matrix, T the temperature field and F the heat source field.

From equation (2.10) it can be noticed that as the number of grid points n increases, the size of the global conductivity matrix K increases by n^2 and thus, for large mesh sizes the computation of the global matrix becomes a major bottleneck in terms of computational time and memory requirements for solving the linear system $KT = F$. For example, in a FEA of the 3D heat conduction on a computational mesh composed of 200 mesh elements in each direction (x,y,z) which leads to $201^3 = 8.12M$ grid points, the size of the global conductivity matrix becomes $8.12M^2 = 65.93$ trillion grid points. Despite the fact that several smart techniques [53] have been developed to avoid storing any zero value instances in the global matrix, the memory requirements for its computation are very large that do not fit in the available resources. In order to achieve finite element simulations with the global matrix assembly method of such big sizes, computational algorithms with distributed memory parallelism are needed like the Message Passing Interface (MPI) [54] software protocol for

the communication and data transfers between the parallel computers. However, the implementation of such parallel algorithms is not a trivial task and additional time-delays are difficult to be avoided for the communication between the successive cores especially when the number of parallel cores are used is very high [55].

Figure (2.2) shows a visual example of how the assembly matrix is implemented on a simple 2D finite element case. In contrast with structural analysis, in heat transfer simulation one degree-of-freedom is accounted to one grid node since temperature is independent of space orientation. Considering the formulation of the nodes for 3 quadrilateral elements, as shown in figure (2.2), the analytical expression of equation (2.10) takes the following matrix form:

$$\left\{ \begin{matrix} e_1 \begin{bmatrix} T_1 \\ T_2 \\ T_5 \\ T_4 \end{bmatrix} \\ e_2 \begin{bmatrix} T_2 \\ T_3 \\ T_6 \\ T_5 \end{bmatrix} \\ e_3 \begin{bmatrix} T_5 \\ T_6 \\ T_8 \\ T_7 \end{bmatrix} \end{matrix} \right\} = \begin{bmatrix} 1 & 0 & 0 & 0 & 0 & 0 & 0 & 0 \\ 0 & 1 & 0 & 0 & 0 & 0 & 0 & 0 \\ 0 & 0 & 0 & 0 & 1 & 0 & 0 & 0 \\ 0 & 0 & 0 & 1 & 0 & 0 & 0 & 0 \\ 0 & 1 & 0 & 0 & 0 & 0 & 0 & 0 \\ 0 & 0 & 1 & 0 & 0 & 0 & 0 & 0 \\ 0 & 0 & 0 & 0 & 0 & 1 & 0 & 0 \\ 0 & 0 & 0 & 0 & 1 & 0 & 0 & 0 \\ 0 & 0 & 0 & 0 & 1 & 0 & 0 & 0 \\ 0 & 0 & 0 & 0 & 0 & 1 & 0 & 0 \\ 0 & 0 & 0 & 0 & 0 & 0 & 0 & 1 \\ 0 & 0 & 0 & 0 & 0 & 0 & 1 & 0 \end{bmatrix} \begin{Bmatrix} T_1 \\ T_2 \\ T_3 \\ T_4 \\ T_5 \\ T_6 \\ T_7 \\ T_8 \end{Bmatrix} \tag{2.11}$$

The procedure described in equation (2.11) is sufficient to form the global matrix assembly and same procedure is also applied to three-dimensional problems. For example, for an equivalent 3D hexahedral mesh with the same structure of the one showing in figure (2) the matrix A for 3 cubic elements would be a [24 x 16] matrix instead of the [12 x 8] for the 2d case and so on.

One interesting point in the formulation of the global assembly matrix is that the degree-of-freedom (DOF) index is transferred from the local element to the corresponding global

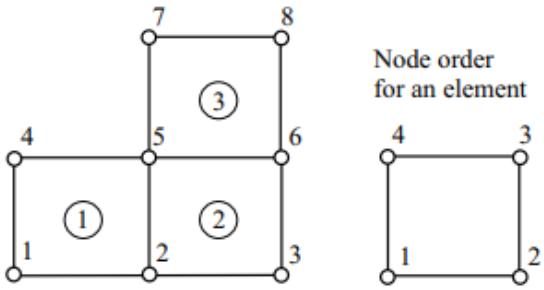


Figure 2.2: A schematic representation of the local enumeration of the nodes of a single element (right) and the global indeces of both the nodes and the elements (left).

position. Understanding the transfers of the local to global and index becomes the key factor for the implementation of the the matrix-free method. Storing and applying those index transfers in the matrix-free method is greatly facilitated by the regular connectivity of the voxel-based grid. This is explained by the voxel-based technique in which the connectivity of the mesh elements and nodes follows a certain order incremented by one until the last mesh node. This regular mesh connectivity implies that for a given DOF in the domain $DOF(index + 1) = DOF(index) + 1$ starting from $index = 0$ until $index = N$ with N the number of mesh elements.

2.2 Neccessity of the Matrix-Free Method

This section presents the matrix-free method as an alternative "implicit" way for solving the equivalent linear system. The term "implicit" means that the algorithm does not store the global conductivity matrix explicitly, but accesses the matrix by evaluating matrix-vector products. The matrix-free method is preferable when the linear system is so big that storing and manipulating the global stiffness matrix it would cost a lot of memory and computer time, even with the use of methods for sparse matrices. Especially, in 3D

industrial-scale simulations, like the thermal simulation of the laser powder bed fusion (L-PBF) process for additive manufacturing (AM), the size of the computational mesh exceeds the order of millions of elements [56]. Sometimes the global system matrix K becomes too large to fit in the available memory of the computer causing a memory bottleneck for the FEA. Furthermore, when solving non-linear problems or problems with time-dependent coefficients, it becomes necessary to re-assemble the system matrix with a higher frequency throughout the simulation. This changes the relative work size of the assembly phase, and precomputing the large system matrix for only a few matrix-vector products may not be efficient. Similarly, when using adaptive mesh refinement, the matrix has to be re-assembled each time the mesh is changed. For all the afformentioned reasons it can be observed that storing the global stiffness matrix becomes a bottleneck for solving the finite element system of equations.

2.3 Introduction of the Matrix-Free Method

Iteratively solving the large sparse linear equation system $KT = F$ globally is omitted with the application of matrix-free methods. Generally the equation is transformed for iterative methods like the conjugate gradient method such that

$$r = F - KT \tag{2.12}$$

where r is the residual vector. The matrix-vector product, which is the most time-consuming part of the solution algorithm [10], is rewritten by introducing the source src and the destination dst vectors.

$$dst = K\ src \tag{2.13}$$

where dst is the result (destination) vector for the inner step of the matrix-vector product and src is the search direction (source) vector calculated appropriately by the applied iterative method. The Eq. (2.21) is then computed with matrix-free methods. In the next section (2.2.3) the algorithm of the matrix-free method is presented considering an underlying uniform structured grid model.

2.4 Algorithm of the Matrix-Free Method

One benefit of the regular grid discretization with elements of the same size, orientation and shape is the simplified topology, which allows a substantially reduced data structure with minimal memory requirement. Furthermore, since isotropic element behavior is assumed on the heat conduction only one element conductivity matrix with unit value of thermal conductovity $\kappa = 1$ is needed to be stored and computed throughout the simulation. In contrast to the strategy presented in [10] where all element stiffness matrices are stored, the proposed matrix-free method reduces the required main memory substantially. Therewith huge models are feasible on modern desktop computers.

On this basis, the linear equation system is solved with local matrix-vector calculations, without calculation or storage of the global conductivity matrix (matrix-free) and this way the computational effort is reduced as well as the needed memory demand. The remaining local matrix-vector calculations are of the order of the numbers of degrees-of-freedom (DOFS) of one element. Therefore, for an 8-node hexahedral element conductivity matrix is an $[8 \times 8]$ matrix for a thermal analysis and a stiffness element $[24 \times 24]$ matrix for a structural analysis system. Since the numerical experiments in this thesis are performed for a the steady-state thermal conduction FEA, the number of matrix-vector operations for each mesh element is $[8 \times 8]$ such that each node position at a given row i and column j of the element conductivity matrix K_{ij} is multiplied by the corresponding source vector value src_j for a node of the hexahedral element resulting to a value at the node position of a destination vector dst_i.

The result vector dst is calculated once per iteration step within the algorithm 2. numElems is the total number of finite elements and the index e corresponds to the element id. For each element the local degrees-of-freedom (DOFs) of the search direction vector src are collected and then multiplied with the local conductivity matrix K_e. Finally the local solution dst_e is assembled back in the global vector dst. To this end, it can be proved that the global matrix assembly is not necessarily needed to be stored. In order to explain the matrix-free method with a simple example the 1D expression of the linear system $KT = F$ is described. It is assumed in the figure (2.3) the four 2-node linear finite elements and 5 nodes with identical element conductivity matrices with unit thermal conductivity.

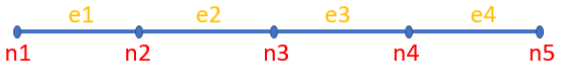


Figure 2.3: A schematic representation of 4 one-dimensional bar elements.

$$\begin{bmatrix} 1 & -1 & 0 & 0 & 0 \\ -1 & 1+1 & -1 & 0 & 0 \\ 0 & -1 & 1+1 & -1 & 0 \\ 0 & 0 & -1 & 1+1 & -1 \\ 0 & 0 & 0 & -1 & 1 \end{bmatrix} \begin{bmatrix} T1 \\ T2 \\ T3 \\ T4 \\ T5 \end{bmatrix} = \begin{bmatrix} F1 \\ F2 \\ F3 \\ F4 \\ F5 \end{bmatrix} \tag{2.14}$$

Indeed, the procedure for the matrix-free algorithm for a node that is shared in neighboring elements, (e.g node 2) proves that the global assembly of the conductivity matrix $K = \sum_e K_e$ is not neccessary to be implemented and stored. The following equation shows the calculation of the matrix-vector product on the element level for the node n_2, which is shared between the finite elements e_1 and e_2 without assembling the global matrix. The result of the matrix-free matrix-vector product is stored as the variable F_2 on the right hand side.

$$K_e^{e_1}[2][1]T[1] + K_e^{e_1}[2][2]T[2] + K_e^{e_2}[1][1]T[2] + K_e^{e_2}[1][2]T[3] = F[2], \tag{2.15}$$

where $K_e^{e_1}[2][1] = -1$, $K_e^{e_1}[2][2] = 1$, $K_e^{e_2}[1][1] = 1$ and $K_e^{e_2}[1][2] = -1$.

The matrix-free matrix vector multiplication is described in the algorithm 1. The relationship between the source and destination vectors is $src = \sum_e K_e\, dst$. The variable tmp is a temporal variable that stores the sum of the matrix-vector products. The integer $numElems$ is the total number of element on the structured mesh, $numrows$ is the number of rows of the element conductivity matrix and $numnodes$ the number of nodes of the hexahedral element.

Algorithm 1 Algorithm of the matrix-free matrix vector product **MFMV(dst,src)**

set $dst = 0$
for $e = 1 \rightarrow numElems$ do
  for $row = 1 \rightarrow numrows$ do
    extract row_{index}
    set $tmp = 0$
    for $node = 1 \rightarrow numnods$ do
      extract dof_{index}
      $tmp = tmp + K_e \cdot src$
    end
    $dst = dst + tmp$
  end
end
return $\rightarrow dst$

3.0 Matrix-Free Conjugate Gradient in Finite Element Analysis

3.1 Problem Statement

The steady-state heat conduction equation for an isotropic body with unit thermal conductivity can be described as

$$-\left(\frac{\partial^2 T}{\partial x^2}+\frac{\partial^2 T}{\partial y^2}+\frac{\partial^2 T}{\partial z^2}\right)=Q(x,y,z), \tag{3.1}$$

where on the left hand side is the Laplacian second partial derivatives of the temperature and on the right hand side $Q(x,y,z)$ is the constant heat source at x, y, and z space. The negative sign on the heat fluxes describes the direction of the thermal energy from hotter to colder temperatures.

3.2 Computational Domain - Boundary Conditions

The computatinal domain for the heat transfer simulation is a uniform structured 3D mesh composed of equal size hexahedral cubic 8-node elemens. This means the number of elements in all x, y and z direction of the mesh is the same. For example, if the number of elements in direction x is N, then the total mesh size is N^3 elements. The nodes are following the same index incremented by one such that $numNodes = (numElems + 1)$ in each direction. Hence, if N^3 is the number of total elements on the mesh, then the number of total grid nodes is $(N+1)^3$. Dirichlet type of boundary condition $T = 0$ are imposed on all 6 boundary surfaces of the domain. Figure (3.1) shows a visual representation of the computational domain with its 6 boundary faces or bases of the box [bottom, top, left, right, front, back].

The time point in the main program where the boundary conditions are imposed is just after the matrix-vector operation, which is described in the algorithm 1. The algorithm for the Dirichlet boundary conditions is presented in algorithm 2. As it is described in Algorithm

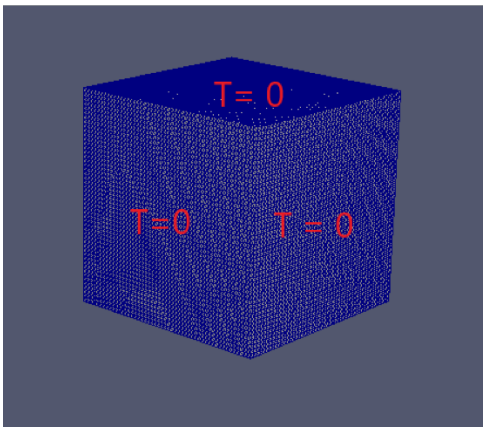


Figure 3.1: A schematic representation of the 3D hexahedral mesh and its 6 boundary surfaces.

1, the relationship between source src and dst vector is $src = \sum_e K_e \cdot dst$. The Dirichlet Boundary condition is specified with zero values in the source and destination vectors for the nodes that are lying on the 6 boundary faces of the computational domain. Therefore, for the initialization of the linear system $KT = F$, the boundary conditions are $T_{bf} = F_{bf} = 0$ with T_{bf} and F_{bf} the corresponding values of T and F on the boundary faces.

Algorithm 2 Dirichlet boundary conditions on the 6 faces of the domain **DirichletBC**

1: for $e = 1 \rightarrow$ numElem do
2: $src \rightarrow 0 \ \forall \, e \ \in \ boundary \ face$
3: $dst \rightarrow 0 \ \forall \, e \ \in \ boundary \ face$
4: end

3.3 Initialization of Solution and Heat Source

For the beginning of the numerical computations, the numerical solver of the linear system $KT = F$, except from the computation of the element conductivity matrix K requires the initialization of the temperature vector T with an initial guess value, and the heat source term which is defined as the right hand side (RHS) vector F. We assume the simplest case of itiliazitation with zero temperature $T = 0$ and $F = 10^{-5}$ constant value of heat source. The solver is initiated with the calculation of the residual vector r by transfering all the terms to the left hand side (LHS) and solving for the residual: $r = F - KT$. The numerical solution is being updated in each iteration by updating the values of the residual vector and re-forming the system using a technique of minimizing the residual norm. The algorithm of the matrix-free conjugate gradient solver that is used for the computation of the numerical solution of the temperature is described in the next section 3.4.

3.4 The Matrix-Free Conjugate Gradient Method

Large sparse systems often arise when numerically solving partial differential equations or optimization problems. The conjugate gradient (CG) method is a well-known iterative-based method, which efficiently solves the large and sparse general linear equation system $KQ = B$ with a symmetric positive definite matrix $K \in \mathbb{R}^{n\times n}$, the solution vector Q (e.g displacement, temperature) and the force or source vector B at the right hand side. It can be mathematically expressed as a minimization problem of the objective function of the unknown vector Q and assuming exact arithmetic, converges in at most n steps, where n is the size of the matrix of the system. It was mainly developed by Magnus Hestenes and Eduard Stiefel,[50] [51] who programmed it on the Z4 [52] the world's first commercial digital computer.

In this thesis, the solution of the steady-state distribution of the temperature inside a bounded equal-sized cubic domain is calculated by minimizing the objective function of the temperature vector. The linear system of equations takes the form $KT = F$ with K

the conductivity matrix, T is the temperature field and F is the heat source vector. The objective function takes the form

$$obj(T) = \frac{1}{2} T^{\intercal} K T - T^{\intercal} F \tag{3.2}$$

and is minimized if

$$T = \widehat{min} \left\{ obj(T) \right\} \text{ for } KT - F = 0 \tag{3.3}$$

The minimum of the function $obj(T)$ is found successively following special types of search directions. In the CG method the residual vectors are used to generate search directions. In each iteration step the matrix-vector multiplication spans a new subspace where the new search direction is K-orthogonal and the new residual is orthogonal to all previous search directions and gradients. The CG method combines the positive properties of the method of steepest descent (problem orientated) and conjugate directions (optimized). The method converges theoretically to the exact solution after n iteration steps. Considering round-up errors for the numerical solution the iterative process is stopped after reaching a sufficient small tolerance of the residual norm (e.g $r_{norm} = 10^{-6}$). The value of the residual norm under the tolerance values inform the solver to stop the iterations and accept the solution as it is stored in the last iteration step because it indicates that the values of the temperature vector are not changing up to a significant digit or equivalently the temperature gradients tend to be zero. In this work the precision or accuracy of the computations of the CG solver is of the order of $\epsilon = 10^{-6}$.

Algorithm 3 describes the procedure for the implementation of the matrix-free conjugate gradient (MFCG) solver of the solution of the linear system $KT = F$, with T the unknown temperature field, K the [8 x 8] sparse symmetric positive definite element conductivity matrix and F the heat source vector. The vectors d and h are introduced in the MFCG algorithm to store and update the resulting values in each iteration. In the MFCG algorithm ϕ is the linear search direction, α is the conjugate direction search and β is the gradient of line search ϕ in each iteration for updating the solution vector T.

The MFCG method is efficient for sparse quadratic systems and it converges after n finite steps. It is actually an extension of the steepest descent gradient method where the only direction search is based on the residual gradient. The term "conjugate" denotes that the

minimization direction of the quadratic solution $obj(T)$ searches in the conjugate direction which is the square root of the horizontal plus the vertical line searches in the steepest descent (SD) method. A geometric view of the search directions of such solvers can be seen in figure (3.2) Ref. [61].

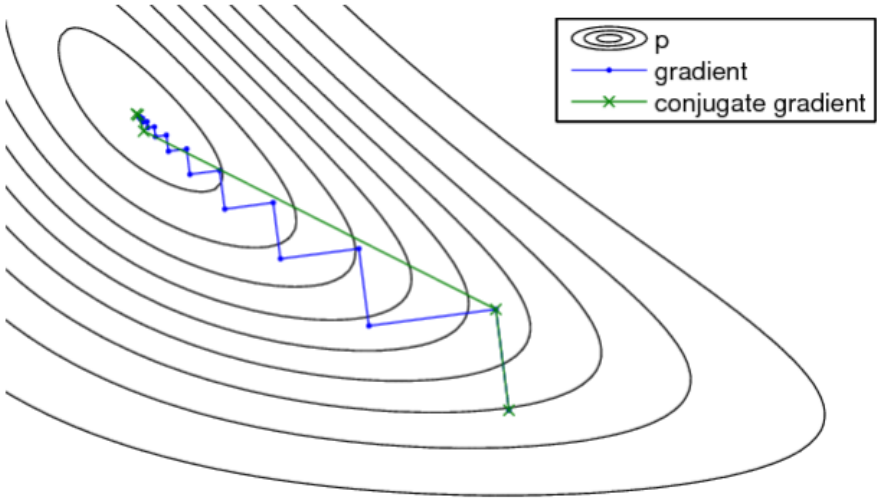


Figure 3.2: A schematic representation of the rate of convergence of the conjugate gradient (green) and simple gradient (blue) methods.

Algorithm 3 Algorithm of the MFCG method

1: Initialize solution with initial guess $T \rightarrow 0 \ \forall \, nodes$

2: Call $MF - SpMV(r_0, T)$

3: $r = r_0 - F \ \forall \, nodes$

4: $dst = r \ \forall \, nodes$

5: $\phi_{old} = r^{\intercal} r$

6: Set residual norm $r_{norm} \rightarrow \ 0$

7: Set tolerance $\epsilon \rightarrow \ 0$

8: Set $Maxiter = 100000000$

9: for $iter = 0$ until Maxiter do

10: $\quad src \rightarrow 0$

11: $\quad dst \rightarrow 0$

12: $\quad Call \ MF - SpMV(dst, src)$

13: $\quad h = h + src * dst$

14: $\quad \alpha = \frac{\phi_{old}}{h}$

15: $\quad T = T + \alpha * src$

16: $\quad r = r - \alpha * dst$

17: $\quad r_{norm} = \sqrt{r^T * r}^2$

18: $\quad if \ (r_{norm} \ \leq \ \epsilon)$ break;

19: $\quad \phi_{new} = 0$

20: $\quad \phi_{new} = r^{\intercal} r$

21: $\quad \beta = \frac{\phi_{new}}{\phi_{old}}$

22: $\quad src = r + \beta * src$

23: $\quad \phi_{old} = \phi_{new}$

24: $\quad iter = iter + 1$

25: end

4.0 High Performance Computing for Finite Element Analysis on Graphics Processing Units

4.1 Background

As widely known, computer processors undergo an extremely fast development. According to the empirical observation of Moores law, the number of transistors in a chip grows exponentially with a doubling every two years [62]. However, since the early 2000s this does no longer translate directly into a corresponding increase in serial performance [63]. At that point, it was no longer possible to make significant increases in the clock frequency, since voltage could no longer be scaled down with the transistor size due to power issues. This is usually referred to as the power wall, or breakdown of Dennard scaling [64]. Instead, the focus has shifted towards increasing parallelism within the processor in the form of multicore designs like the known graphical processor unit (GPU) architectures. This change has increased the burden on the programmer since utilizing a parallel processor is more complex than a serial one. When writing a parallel program, the work must be split into smaller tasks that can be performed concurrently, and issues such as communication and synchronization between parallel tasks must be addressed.

4.2 Techniques for Handling Race Conditions

One of the main problems when programming multicore processors (GPUs) is the coordination of memory access. In contrast to the cluster computers where the memory is distributed over the nodes, a multicore processor or GPU device has a single memory which is shared between the cores. The GPU tensor cores are typically programmed using threads, which all have access to the shared memory. While flexible, this approach can lead to faulty operations called “race conditions“. A “race condition“ occurs when threads concurrently manipulate the same memory location [14].

Table 4.1: Two threads increment the same variable x concurrently, leading to the result $x = 1$ instead of the expected $x = 2$.

x	**thread 1**	**thread 2**
0	$x_1 \leftarrow x$	
0	$x_1 \leftarrow x_1 + 1$	$x_2 \leftarrow x$
1	$x \leftarrow x_1$	$x_2 \leftarrow x_2 + 1$
1		$x \leftarrow x_2$

It is up to the application programmer or library developer to guarantee that no race conditions can appear, and this is one of the main difficulties when writing multithreaded programs. The code segments that potentially might conflict with each other, or with other copies of itself, is referred to as critical sections. In order to avoid race conditions, the critical sections need to be executed mutually exclusively, i.e. only a single thread is allowed within a critical section simultaneously [65].

The most common technique for achieving exclusivity is to use locks, which are mutually exclusive data structures with two operations lock, which obtains the lock, and unlock, which releases the lock [66]. If the critical section is surrounded with a lock and an unlock operation, then only a single thread will be allowed to be in the critical section, since any other thread will not succeed with the lock operation until the first thread has completed the unlock operation following its critical section.

Another way of achieving mutual exclusivity is the concept of atomicity [66]. If all the operations in the critical section are considered an indivisible entity, which can only appear to the memory as a whole, then it can be executed safely concurrently. For simple critical sections, e.g. an incrementation of a variable, the processor architecture may offer native atomic instructions. Atomic instructions usually perform well because of the efficient implementation in hardware. However, they cannot be used for general critical section and are limited to the available atomic instructions available. Although simple atomic instructions

such as compare-and-swap can be used to implement somewhat more complex atomic operations, this is still very limited since, typically, not more than a single memory location can be manipulated.

Locks on the other hand are completely general, but can lead to several performance related issues. A deadlock occurs when two or more threads are waiting for each others locks [68]. Lock convoying is when many threads are waiting for a lock while the thread holding the lock is context switched and prevented from progressing. Priority inversion happens when a low-priority thread holds the lock, preventing execution of threads of higher priority.

Another more recent technique which also uses atomicity to achieve mutual exclusion is transactional memory [67]. Rather than surrounding the critical section by lock and unlock operations, all the instructions of the critical section are declared to constitute an atomic transaction. Then, when the transaction is executed, the transactional memory system monitors whether any conflicting operations have been performed during the transaction. If conflicts are detected, the transaction is aborted, i.e., all of its changes to the memory system are rolled back to the pre-transactional state. On the other, if no conflicts were detected, the transaction commits, i.e., its changes are made permanent in the memory system. Since this approach assumes a successful execution, and only deals with conflicts if they appear, it can be regarded as an optimistic approach. This is in contrast to the locks-based technique, where we always perform the locking even if no actual conflicts occurred, thus being a more pessimistic approach. This can potentially lead to lower overheads for the cases when contention is low, i.e., when conflicts are rare.

4.3 Computational Capability and Precision Architecture Trends on GPUs

One basic computer measurement of a GPU performance is the floating point operations per second (FLOPS, flops or flop/s). Floating-point arithmetic is needed for very large or very small real numbers, or computations that require a large dynamic range. Floating-point representation is similar to scientific notation, except everything is carried out in base two, rather than base ten. The encoding scheme stores the sign, the exponent (in base two

for Cray and VAX, base two or ten for IEEE floating point formats, and base 16 for IBM Floating Point Architecture) and the significant number (number after the radix point).

FLOPS on an HPC system can be calculated using this equation:

$$FLOPS = \frac{sockets}{node} \times \frac{cores}{socket} \times \frac{cycles}{second} \times \frac{FLOPS}{cycle} \tag{4.1}$$

Another important hardware property of a GPU is its capability for double presicion operations. GPUs tend to have very different performance on double-precision computing. A few GPUs (e.g., the NVIDIA Tesla V100) have much higher performance than CPUs, while other GPUs struggle to be faster than CPUs in double-precision computing. GPUs have very different double-precision computing capabilities because GPU vendors design GPUs for different markets. For gaming-oriented GPUs, the ratio between the number of single-precision units and double-precision units is usually 32:1, according to Ref [69]. For high-performance computing-oriented GPUs, the ratio is usually 2:1, [69]. Interestingly, for GPUs of the same product series, the ratio is not fixed. In recent years, with the development of new CPU technologies, CPUs can have a much higher double-precision performance than many GPUs. It is suggested that users check the specifications of their CPUs and GPUs before using GPUs for double-precision computing.

Some of the most powerful GPU architectures today deliver orders of TFLOPS with ultimate intensity. According to the graph on figure (6) adapted from [69] NVIDIA's TITAN V seems to lead the trends with an approximation of 17 TFLOPS maximum performance on single precison and about 7.5 TFLOPS using double-precision computing while TESLA GPUs seem to perform slightly better with a maximum of almoast 8 TFLOPS for double precision.

4.4 Parallel Matrix-Free Conjugate Gradient on GPU

Parallel conjugate gradient method for execution on a GPU device is implemented using the CUDA computer platform and is achieved with minor changes to the original serial C

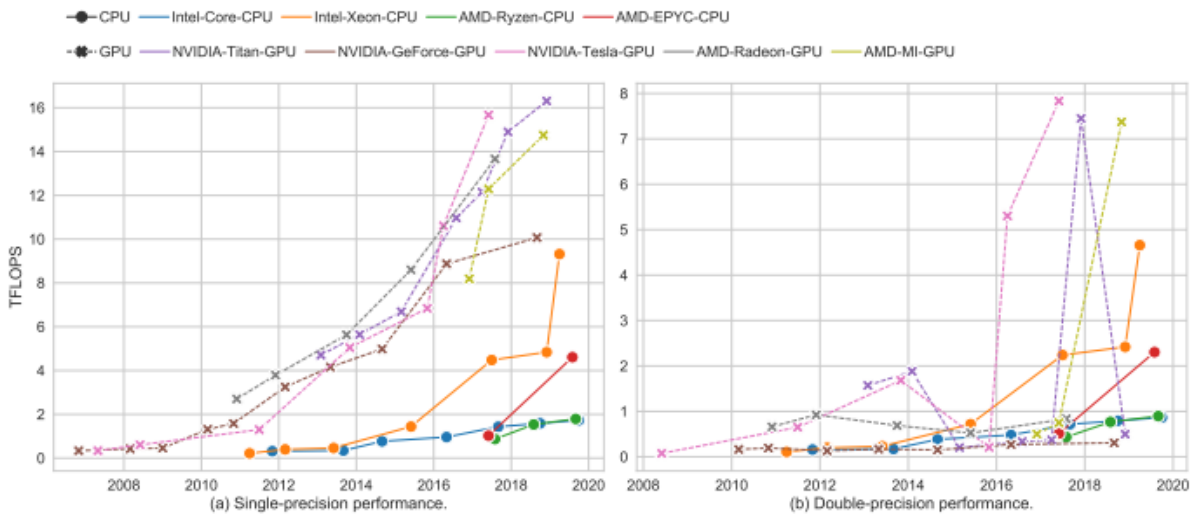


Figure 4.1: Comparison of single-precision and double-precision performance of CPUs and GPUs over the past 10 years, Ref [69].

version of the code that we described in the section 4. The algorithmic steps for the GPU matrix-free congugate gradient (MFCG) are:

$$
\begin{cases}
\textbf{a} : \text{Get GPU Device} \rightarrow \text{cudaSetDevice(gpuID)} \\
\textbf{b} : \text{Allocate GPU Memory for the vectors} \rightarrow \text{cudaMallocManaged}(v_1, v_2, ...) \\
\textbf{c} : \text{Copy host vectors to device} \rightarrow \text{cudaMemcpy}(v_1, v_2, ...)\ \text{cudaMemcpyHostToDevice} \\
\textbf{d} : \text{Perform the MFMV kernel} \rightarrow <<< ELEBLOCKS, BLOCKSIZE >>> \\
\textbf{d} : \text{Perform the DirichletBC kernel} \rightarrow <<< ELEBLOCKS, BLOCKSIZE >>> \\
\textbf{e} : \text{Utilize cuBLAS operations [cublasDdot, cublasDaxpy, cublasDnrm2]} \\
\textbf{f} : \text{Perform updateVec kernel} \rightarrow <<< NODBLOCKS, BLOCKSIZE
\end{cases}
\tag{4.2}
$$

The $BLOCKSIZE$ for all the GPU kernels is chosen 256 threads per block since it is typically considered for optimal acceleration. The $ELEBLOCKS$ is the number of blocks for the mesh element size as $ELEBLOCKS = numElems/BLOCKSIZE + 1$ to account for mesh sizes that are not exactly divisible by the block dimension. Similarly, $NODBLOCKS = numNodes/BLOCKSIZE + 1$ is the number of the blocks for the mesh nodes. This number of blocks is used only in the last kernel when the updated vector of the solution is implemented. The matrix-free matrix-vector product (MFMV) and the dirichlet boundary conditions (DirichletBC) kernels are performed using the $ELEBLOCKS$ for the element-by-element order as it has been discussed in the serial version of the MFCG solver.

The most challenging and time-consuming part of the parallel MFCG algorithm is the implementation of the matrix-free matrix vector kernel on the global device memory of the GPU. The difficulty lies on the fact that during the matrix-free matrix vector (MFMV) algorithm, concurrent values of parallel threads have to be stored and updated at the same time. This type of operation is called "race conditions" leading to the confusion of the memory position of the values where the result vector is updated and the solver diverges very fast. The part of the algorith which faces the race conditions is exactly at the point where the destination vector *dst* is updated with the accumulative summations of the temporal variable *tmp* which in turn stores the sum of the matrix-vector product of the element conductivity matrix K_e with the source vector *src*.

Guided by the CUDA Toolkit documentation [48] the solution to this operation arrived by the use of the CUDA *atomicAdd* function. *atomicAdd* function can handle any "race conditions" might occur while performing vector incremental operations on GPUs and its usage can return different values depending on its exact implementation on the code. For example, if a vector $\vec{\nu}$ needs to be incremented with a scalar variable ω, there are two ways to perform the cudaAtomic operation. To explain this consider the increment of a vector $\vec{\nu}$ with a scalar variable ω n times. The cudaAtomic can be implemented as

$$\begin{cases} \mathbf{1} : \vec{\nu} = atomicAdd(\&\vec{\nu}, \omega) \rightarrow \text{returns the value of } \vec{\nu} \text{ at the step n-1} \\ \mathbf{2} : \text{atomicAdd}(\&\vec{\nu}, \omega) \rightarrow \text{returns the value of } \vec{\nu} \text{ at the last step n} \end{cases} \tag{4.3}$$

In the case of the parallel matrix-free matrix vector (MFMV) algorithm 4, the required usage is the second one which returns the updated value of the result vector after all steps have been implemented. The parallel MFMV algorithm is described on the algorithm 4.

Here it is worth mentioning the optimized Amgx [48] CUDA solver developed by NVIDIA in 2014. It provides a simple path to accelerated core solver technology on NVIDIA GPUs using a global matrix-assembly multigrid algebraic approach. Amgx is a high performance, state-of-the-art library and includes a flexible solver composition system that allows a user to easily construct complex nested solvers and preconditioners. It provides up to $10 \times speed-up$ to the computationally intense linear solver portion of simulations, and is especially well suited for implicit unstructured methods.

Algorithm 4 Parallel matrix-free matrix vector product **MFMV**(dst, src)

1: e = blockIdx.x $\cdot$ blockDim.x + threadIdx.x
2: if $(e < numElems)$ do
3: for $row = 1 \rightarrow numrows$ do
4: extract row_{index}
5: for $node = 1 \rightarrow numnods$ do
6: extract dof_{index}
7: $tmp[node] = K_e \cdot src$
8: end
9: $sum = tmp[node]$
10: atomicAdd$(\&r[dof_{index}[row]], sum)$
11: end
12: end
13: return $\rightarrow dst$

5.0 Numerical Simulation of the Steady-State Heat Transfer

The numerical experiments conducted to test the performance of the matrix-free conjugate gradient method compared to open source global matrix assembly-based solvers. Specifically, a benchmark analysis is performed (a) between a serial in house global-matrix assembly solver using the PETSc library and the CPU version of the MFCG solver and (b) between NVIDIA Amgx and the parallel-GPU MFCG solver on a TITAN V GPU.

5.1 NVIDIA GPU Hardware Specifications

NVIDIA TITAN V is considered one of the most powerful GPUs and is chosen for the computations of the numerical solution mainly because of its good rate between single and double precision operations. TITAN V has a power of 12 GB HBM2 memory and 640 Tensor Cores, achieving 110 teraflops of maximum performance according to NVIDIA specifications. It features Volta-optimized NVIDIA CUDA for maximum results and its current stoke value is 3000$ [71].

Figure 5.1: NVIDIA TITAN V GPU [70].

NVIDIA GPU Specification Comparison				
	Titan V	Titan Xp	GTX Titan X (Maxwell)	GTX Titan
CUDA Cores	5120	3840	3072	2688
Tensor Cores	640	N/A	N/A	N/A
ROPs	96	96	96	48
Core Clock	1200MHz	1485MHz	1000MHz	837MHz
Boost Clock	1455MHz	1582MHz	1075MHz	876MHz
Memory Clock	1.7Gbps HBM2	11.4Gbps GDDR5X	7Gbps GDDR5	6Gbps GDDR5
Memory Bus Width	3072-bit	384-bit	384-bit	384-bit
Memory Bandwidth	653GB/sec	547GB/sec	336GB/sec	288GB/sec
VRAM	12GB	12GB	12GB	6GB
L2 Cache	4.5MB	3MB	3MB	1.5MB
Single Precision	13.8 TFLOPS	12.1 TFLOPS	6.6 TFLOPS	4.7 TFLOPS
Double Precision	6.9 TFLOPS (1/2 rate)	0.38 TFLOPS (1/32 rate)	0.2 TFLOPS (1/32 rate)	1.5 TFLOPS (1/3 rate)

Figure 5.2: Specification details of TITAN relative architectures [71].

5.2 Numerical Simulation Specifications

The numerical solution is computed for the finite element analysis of the steady-state heat conduction over a 3D cube domain using different mesh sizes. For the computation of the solution a hexahedral uniform structured grid is considered with a mesh size of SIZE = $[250 \times 250 \times 250]$ elements. This notation yields $[250 + 1] \times [250 + 1] \times [250 + 1]$ number of nodes on the mesh. The physical dimension of the computational domain is [0 1] in meters both in length, width and height directions.

The simulation starts from zero iteration and is computed until the norm of the residual reaches the tolerance of the order of $\epsilon = 1e - 6$. The overall calculations are computed using double-precision operations since the physics of the problem accommodates computations with very small decimal values of the order of $1e - 8$. The right hand side F of the linear system $KT = F$ represents the heat source in which the constant value of $F = 1e - 5$ is given throughout the simulation. The acceleration and the memory reduction of the MFCG algorithm are expected due to its compute-bound style both on the serial-CPU and parallel-GPU implementations of the method.

The boundary conditions are of Dirichlet type (specified constant value) and they are imposed on the 6 boundary faces of the mesh. In this simulation we assume the simplest case of zero temperature values on all boundary faces. This method is applied to the specific case of 3D structured grids constructed using the voxel-based technique where the connectivity of the mesh follows a certain sequencial order everywhere, meaning on the node index, row index and surface index.

For coverting the MFCG to cover more complicated geometries with unstructured grids and different types of boundary conditions (e.g Neumann with zero heat flux expressing a temperature difference on two opposing boundary faces) the algorithm of the matrix-free is not sufficient. In such a case the calculation of the connectivity for the local to global index tranformations would be not as straightforward as it is on structured grids with the voxel-based approach since the order of the mesh elements connectivity does not follow a sequencial order. However, it could be possible by adding new matrices for storing the element and node connectivity of the unstructured grid which would further increase the memory demand.

5.3 Validation of the Numerical Results

The accuracy of the matrix-free conjugate gradient method is validated by comparing the numerical results of the steady-state heat conduction FEA with an in-house global matrix-assembly solver. For the comparison the relative error on the numerical solution of the temperature is computed for a same mesh size of 200^3 elements.

$$\mathbf{error} = \sum_{\mathbf{i=1}}^{\mathbf{N}} \left| \frac{\mathbf{GMCG_i} - \mathbf{MFCG_i}}{\mathbf{MFCG_i}} \right|, \tag{5.1}$$

where GMCG is the global-matrix conjugate gradient and MFCG the matrix-free conjugate gradient method. The value of the relative error is $\mathbf{error} = \mathbf{0.0532}\%$ and confirms the accuracy of the MFCG solver. The numerical results of the matrix-free method are also

verified with the results of the FreeFemm++ [49] library by plotting the countours of the xy, xz and xz planes for both solvers for the same mesh size of 20^3 elements. From figure (5.3) it can be observed that all the profiles on both solvers are showing the same heat conduction distribution and same temperature values in the range [0 - 0.56]. The fact that all the contour profiles have the same shape can be explained by the fact that the computational mesh used for the calculation of the solution is symmetric in all x,y and z directions. The results are also verified for the parallel matrix- free conjugate gradient on GPU returning the same temperature distribution profiles and a relative temperature error of the order of the serial CPU version.

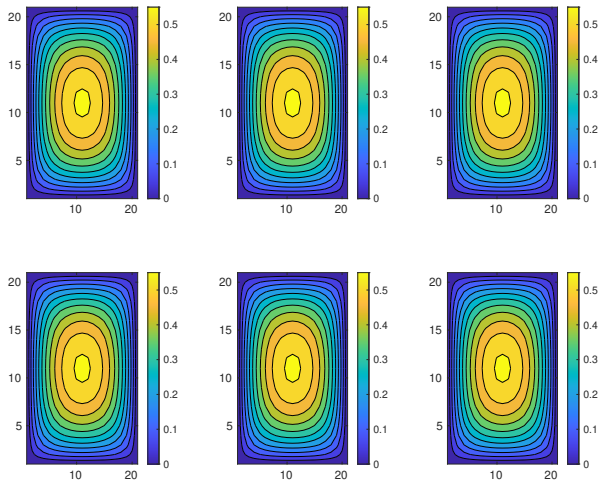


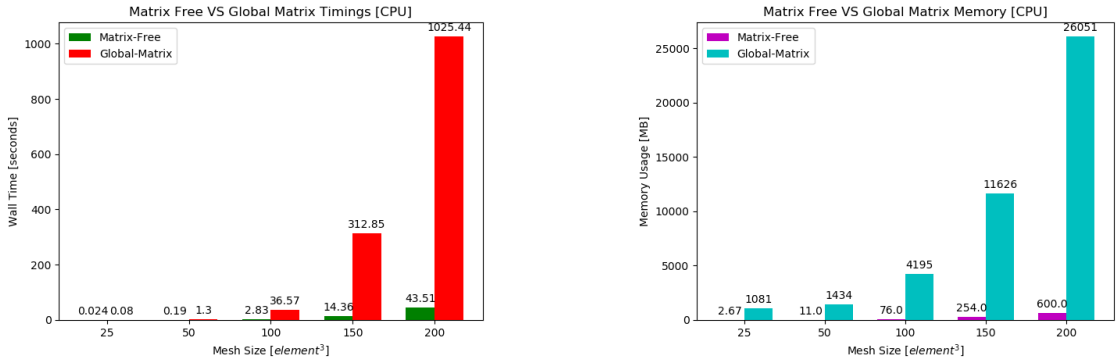
Figure 5.3: Temperature distribution on xy, xz and xz cross sections of a mesh size 20^3: FreeFem++ (top) and MFCG (bottom)

5.4 Performance Analysis of the Serial Matrix-Free Conjugate Gradient

This section presents a benchmark analysis between the matrix-free and global-matrix assembly methods to the finite element computations on the 3D conduction heat transfer simulation. For both algorithms the conjugate gradient method is used to solve the linear

system $KT = F$ on different sizes of 3D hexahedral grids. The analysis evaluates the performance of the two methods on the computational time and the memory usage consumption.

Figure (5.3) highlights the superiority of the matrix-free algorithm both in computational time and memory usage. It is noticed that for small size grids - up to 100^3 elements the two methods show very similar performance in time. On the other side, memory increase of the global matrix assemby is evident even for small sizes. For sizes larger than 100^3 both the computational and the memory usage of the global-matrix assembly are significantly increased. In the case of the 200^3 elements the memory of the global matrix assembly exceeds the value of 26 [GB] memory which is a major bottleneck in large-scale FEA simulations. Even though, the global matrix assembly solver has been designed to store only the non-zero valures of the sparse matrix coefficients the problem is still requires huge amounts of memory that slow-down the computations dramatically. On the contrary, the matrix-free method performs very efficiently requiring less than 44 seconds and 600 [MB] memory for the same mesh size. The speed up of the matix-free over the global-assembly matrix is illustated on the figure (5.4).



(a) Computational Time [SEC]

(b) Memory Usage [MB]

Figure 5.4: Performance of the matrix-free vs global matrix-assembly method

For larger than 200^3 mesh sizes the global-matrix assembly algorithm requires much greater percentages of computational time and memory and thus, the numerical results are

considered only for the matrix-free method. From figure (5.5) it can be inferred that the matrix-free solver shows efficient scaling both in computational time and memory for mesh sizes up to 500^3 mesh elements.

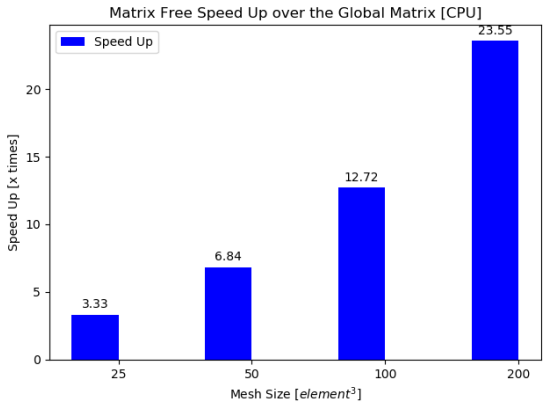


Figure 5.5: Speed-up of matrix-free over the global-matrix assembly method.

The outcome of this performance study of the two methods can be concluded in two arguments:

- For all mesh sizes the matrix-free outperforms the global matrix assembly method.
- Matrix-free shows good scaling performance allowing for numerical solutions of much larger domains.

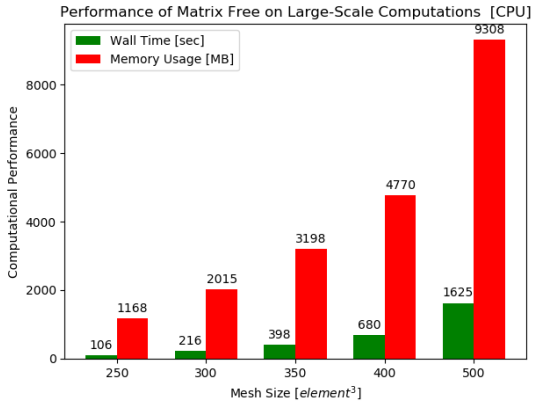
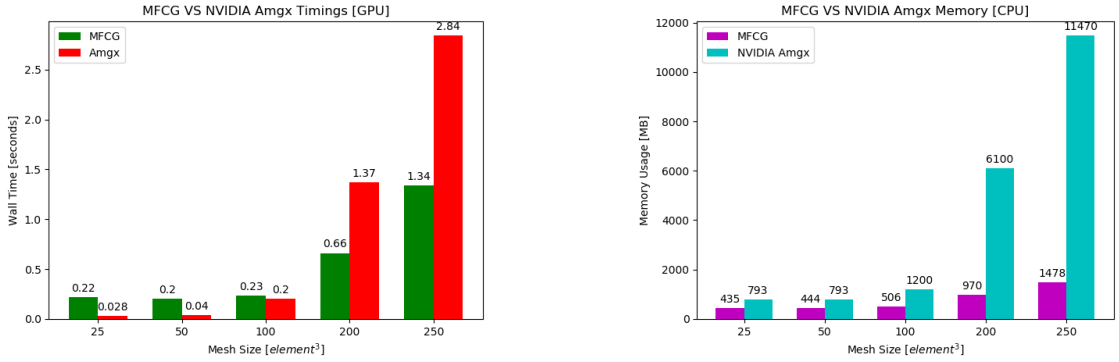


Figure 5.6: Performance of matrix-free method on large-scale simulations.

5.5 Performance Analysis of the Parallel Matrix-Free Conjugate Gradient

In this work, the computer workstation of the Modeling & Optimization Simulation Tools for Additive Manufacturing Laboratory (MOST-AM) is used. The workstation operates on Linux Ubuntu 18.04-LTS and is equipped with 4 slots of TITAN V NVIDIA GPU cards and 8 double-socket Intel(R) cores. The numerical simulation is executed on a single TITAN V GPU (figure 4.5) using double precision operations for maintaining lower numerical rounding error percentages. Later work will be focused on a multi-GPU implementation of the MFCG method utilizing all the available GPUs scaling up to $(550 \times 4)^3$ computational mesh size.

Similarly to the section (5.4), a performance analysis is presented for the matrix-free conjugate gradient solver versus NVIDIA's optimized Amgx, which is a global-matrix assembly type solver. For all the numerical experiments the block dimension of the GPUs is fixed at $BLOCKSIZE = 256$ threads, which is typically considered to achieve optimal acceleration.



(a) Computational Time [seconds]

(b) Memory Usage [MB]

Figure 5.7: Parallel performance of the matrix-free vs NVIDIA's Amgx assembly-based solver

Looking at figure (5.6) it is interested that for mesh sizes up to 100^3 elements the computational time performance of the Amgx open source solver of NVIDIA is slightly better than the developed MFCG solver while in memory usage Amgx consumes more than double of the memory ammount that MFCG does. For large mesh sizes greater than 100^3, the performance of MFCG on both computational time and memory usage is much better. In

particular, the time of the MFCG is constantly half of that of the Amgx solver. Even greater difference is observed on the memory usage where the parallel MFCG solver seems to require 6 and 7 times less memory as the size of the probem increases. Specifically, the mesh size of 250^3 is the maximum memory limit for the Amgx solver whereas the respective limit of the MFCG solver (figure 5.7) is the computational domain of 550^3 hexahedral elements. The numerical results of figure (5.7) highlight the high efficiency of the parallel MFCG for the computation of the numerical simulation of the 3D heat transfer on TITAN V.

The sharp increase in the computational time is noticed for mesh sizes larger or equal to 600^3 mesh elements. This is investigated only for boosting the GPU limits on out of memory operations. This condition is usually slows dramatically the computations on the GPU as there is not sufficient space to save and load the results. This condition is not recommended and it just used only once for testing the performance of the GPU in overrun conditions.

The memory usage of the matrix-free method shows a linear profile with an average slope of 1.3 from size to size in the range [1.50 1.12]. This scaling ability of the matrix-free method is very advantageous over other global matrix assembly-based techniques, especially in realistic industrial-scale simulations where the number of DOFs of the simulation model reaches the order of billion of elements.

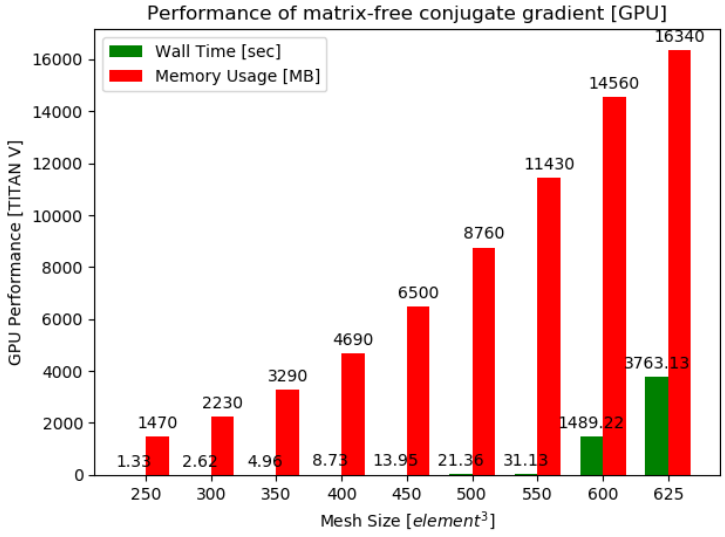


Figure 5.8: Parallel performance of MFCG solver on large-scale simulations.

In figure (5.8) it is also illustrated the speed-up gain of the matrix-free conjugate gradient on TITAN V GPU over its serial version. The maximum acceleration of the parallel MFCG solver is $ACC_{max} = [\text{times} \times 81.81]$ over its serial performance and is observed for the size of $300^3 = 27M$ hexahedral elements of the structured mesh. This is considerably a high percentage for acceleration of the 3D heat transfer finite element simulations. The main factors accounting for this acceleration are a) the reduced algorithmic complexity (order of DOFs) of the matrix-free matrix vector kernel MFMV, b) the utilization of cuBLAS [72] optimized linear algebra functions (ddot,daxpy,dnrm2) and c) the simplified assumptions for the simulation (uniform mesh, steady-state, Dirichlet BC).

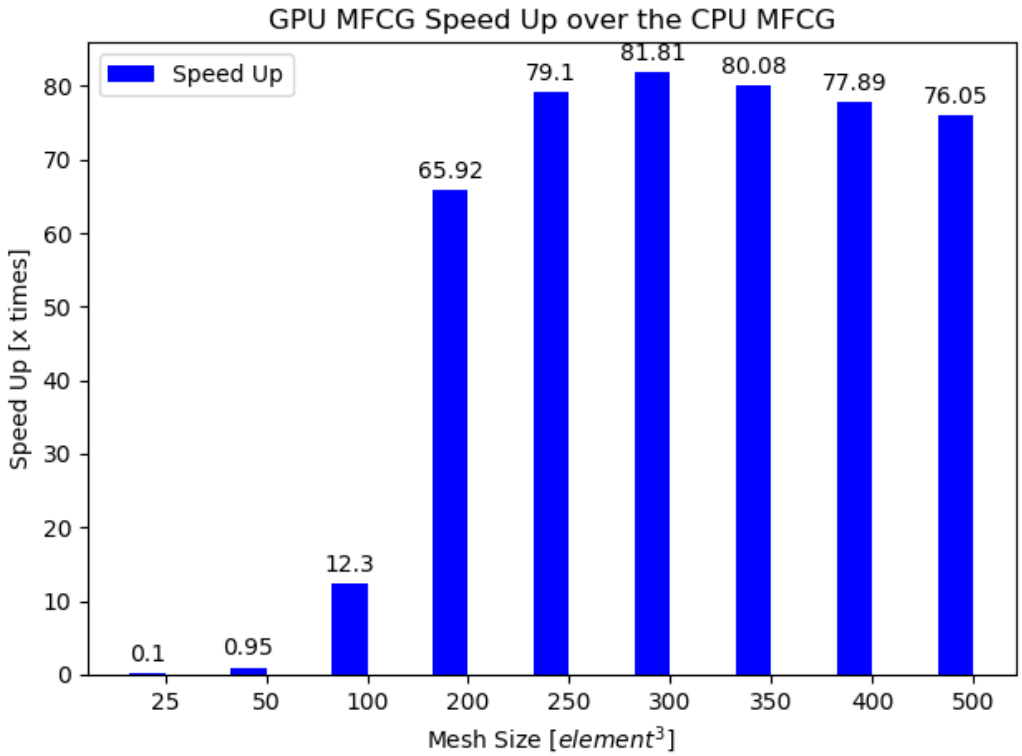


Figure 5.9: Speed-up of the parallel MFCG solver for varying mesh sizes.

6.0 Conclusion and Future Research

6.1 Conclusion

This work investigated the efficiency of a linear iterative solver for large-scale simulations in the field of finite element analysis. The main emphasis is given on the performance of the computations for the heat transfer numerical simulation on large-size uniform structured hexahedral grids. Two types of iterative linear solvers are evaluated. The main difference of the solvers lies on the way they have been programmed to compute the matrix-vector operator during the phase of the solution. On one side, the naive conjugate gradient method is used, which frequently stores and updates the global conductivity matrix during the solution stage. On the other side, an efficient lightweight matrix-free conjugate gradient algorithn (MFCG) is programmed which avoids the global-assembly phase using local element level multiplications and vector superpositions on the global index. The computational performance of the two solvers is tested on the computational time and memory usage requirements on computing the numerical solution of the finite element heat conduction simulation on a uniform structured hexahedral mesh. The comparison of the MFCG and the global-assembly method is performd both for a serial and a parallelized CUDA version utilizing a TITAN V GPU device. For the later case of GPU accelerations the developed MFCG solver is compared with the Amgx open source multigrid solver officially developed by NVIDIA in 2014.

The study of this thesis on the performance analysis of the matrix-free and global-matrix assembly conjugate gradient solvers is summarized on the following foundings:

- The matrix-free conjugate gradient outperforms the global matrix assembly conjugate gradient solver leading to faster large-scale simulations.
- The matrix-free conjugate gradient MFCG requires much lower amounts of memory allowing for numerical simulations of large-scale domains.
- The mesh size limit of the matrix-free conjugate gradient MFCG solver is more than 10.50× times the mesh size limit of the NVIDIA Amgx global assembly-based solver.

- The maximum acceleration of the CUDA parallelized version of the matrix-free conjugate gradient is 81.81× times the CPU version for the mesh size of 300^3 hexahedral elements.
- The memory scalability of the matrix-free conjugate gradient solver has a linear profile of an 1.3 average slope by size.
- Overall, the matrix-free conjugate gradient algorithm by requiring much lower computational time and storage on the matrix-vector multiplication phase performs better than the traditional global matrix assembly conjugate gradient for large-scale finite element analysis.

6.2 Future Research

This thesis is a basic procedure for high performance computing techniques for large-scale finite element analysis simulations. The matrix-free method can be extended through future work to cover geometries with more complex boundary conditions (e.g Neumann BC) and different material properties using a multi-GPU implementation on a single node platform. An example of such application is the layer-by-layer thermo-structural FEA analysis for a build-scale size of $2200^3 = 10.64$ billions finite elements like the laser powder bed fusion (L-PBF) for additive manufacturing (AM). The flexibility of the matrix-free method promises remarkable savings both in computational time and memory usage such that its utilization in an optimization process could lead to fast predictions for additive manufacturing of metal parts with minimized residual stresses and distortions. Another perspective for future work is the developement of a hybrid high performance computing (HPC) implementation of the matrix-free method to run on multiple computer nodes utilizing both distributed memory parallelization on CPUs via inter-node message passing interface (MPI) and multi-node GPU resources. This could lead to additional distribution of the computational domain limiting the size of datas being transferred and computed in each device to a reasonable small level. By this way, the maximum speed-up of each device could be maintaned avoiding memory overruns and thus, large industrial-scale realistic simulations could be perfomed in a feasible time.

Bibliography

[1] Hughes, T., Levit, I., and Winget, J. *"An element-by-element solution algorithm for problems of structural and solid mechanics"*. In: Computer Methods in AppliedMechanics and Engineering 36.2 (1983), pp. 241254.

[2] Coutinho, A., Alves, J., Landau, L., Lima, E., and Ebecken, N. *"On the application of an element-by-element lanczos solver to large offshore structural engineering problems"*. In: Computers & Structures 27.1 (1987), pp. 2737.

[3] Liu, S., Li, X., Wang, W., and Liu, Y. *"A mixed-grid finite element method with PML absorbing boundary conditions for seismic wave modelling"*. In: Journal of Geophysics and Engineering 11.5 (2014).

[4] Zohdi, T. *"Modeling and Simulation of Laser Processing of Particulate-Functionalized Materials"* In: Archives of Computational Methods in Engineering 24.1 (2017),pp. 89113.

[5] Li, W. *"A matrix-free, implicit finite volume lattice Boltzmann method for steady flows"*. In: Computers and Fluids 148 (2017), pp. 157165

[6] Fambri, F. and Dumbser, M. *"Spectral semi-implicit and space-time discontinuous Galerkin methods for the incompressible Navier-Stokes equations on staggered Cartesian grids"*. In: Applied Numerical Mathematics 110 (2016), pp. 4174.

[7] May, D. A., Brown, J., and Le Pourhiet, L. *"A scalable, matrix-free multigrid preconditioner for finite element discretizations of heterogeneous Stokes flow"*. In: Computer Methods in Applied Mechanics and Engineering 290 (2015), pp. 496523.

[8] Gao, S., Habashi, W., Isola, D., Baruzzi, G., and Fossati, M. *"A Jacobian-free Edge-based Galerkin Formulation for Compressible Flows"*. In: Computers and Fluids 143 (2017), pp. 141156.

[9] Kronbichler, M. and Wall, W. *"A performance comparison of continuous and discontinuous Galerkin methods with multigrid solvers, including a new multigrid scheme for HDG"*. In: SIAM Journal on Scientific Computing.2016 (2016).

[10] Ribeiro, F. and Coutinho, A. *"Comparison between element, edge and compressed storage schemes for iterative solutions in finite element analyses"*. In: International Journal for Numerical Methods in Engineering 63.4 (2005), pp. 569588.

[11] Elias, R., Martins, M., and Coutinho, A. *"Parallel edge-based solution of viscoplastic flows with the SUPG/PSPG formulation"*. In: Computational Mechanics 38.4-5 (2006), pp. 365381.

[12] Coutinho, A., Martins, M., Alves, J., Landau, L., and Moraes, A. *"Edge-based finite element techniques for non-linear solid mechanics problems"*. In: International Journal for Numerical Methods in Engineering 50.9 (2001), pp. 20532068

[13] Andrea Keszler *"Matrix-free voxel-based finite element method for materials with heterogeneous microstructures"*. D-Ing dissertation, Bauhaus University, Germany,(2017), pp. 19-25

[14] Karl Ljungkvist *"Techniques for Finite Element Methods on Modern Processors"*. Dissertation for the degree of Licentiate of Philosophy in Scientific Computing, Department of Information Technology, Uppsala University, Sweden (2015).

[15] E. S. Larsen and D. McAllister, *"Fast Matrix Multiplies Using Graphics Hardware"*, in Supercomputing, ACM/IEEE 2001 Conference, pp. 4343, 2001.

[16] J. Kruger and R. Westermann, *"Linear algebra operators for GPU implementation of numerical algorithms"*, ACM Transactions on Graphics, vol. 22, pp. 908916, JUL 2003.

[17] J. Bolz, I. Farmer, E. Grinspun, and P. Schrder, *"Sparse matrix solvers on the GPU: Conjugate gradients and multigrid"*, ACM Transactions on Graphics, vol. 22, pp. 917924, JUL 2003.

[18] J. D. Owens, D. Luebke, N. Govindaraju, M. Harris, J. Krger, A. Lefohn, and T. J. Purcell, *"A Survey of General-Purpose Computation on Graphics Hardware"*, in Eurographics 2005, State of the Art Reports, pp. 2151, 2005.

[19] NVIDIA Corporation, NVIDIA CUDA C Programming Guide, July 2013. Version 5.5.

[20] J. A. Anderson, C. D. Lorenz, and A. Travesset, *"General purpose molecular dynamics simulations fully implemented on graphics processing units"*, Journal of Computational Physics, vol. 227, no. 10, pp. 5342-5359, (2008).

[21] E. Elsen, P. LeGresley, and E. Darve, *"Large calculation of the flow over a hypersonic vehicle using a GPU"* Journal of Computational Physics, vol. 227, no. 24, pp. 1014810161, 2008.

[22] D. Micha and D. Komatitsch, *"Accelerating a three-dimensional finitedifference wave propagation code using GPU graphics cards"*. Geophysical Journal International, vol. 182,no. 1, pp. 389402, 2010.

[23] S. A. Manavski and G. Valle, *"CUDA compatible GPU cards as efficient hardware accelerators for Smith-Waterman sequence alignment"*. BMC Bioinformatics, vol. 9, no. 2, 2008. Annual Meeting of the ItalianSociety-of-Bioinformatics, Naples, ITALY, APR 26-28, 2007.

[24] T. Preis, P. Virnau, W. Paul, and J. J. Schneider, *"GPU accelerated Monte Carlo simulation of the 2D and 3D Ising model"*. Journal of Computational Physics, vol. 228, pp. 44684477, JUL 1 2009.

[25] J. E. Stone, D. Gohara, and G. Shi, *"OpenCL: A Parallel Programming Standard for Heterogeneous Computing Systems"*. Computing in Science & Engineering, vol. 12, pp. 6672, MAY-JUN 2010.

[26] K. Karimi, N. G. Dickson, and F. Hamze, *"A Performance Comparison of CUDA and OpenCL"*. ArXiv e-prints, may 2010.

[27] J. Fang, A. L. Varbanescu, and H. Sips, *"A Comprehensive Performance Comparison of CUDA and OpenCL"*, in Parallel Processing (ICPP), 2011 International Conference on, pp. 216225, Sept 2011.

[28] Z. Fu, T.J. Lewis, R.M. Kirby, R.T. Whitaker, *"Architecting the Finite Element Method pipeline for the GPU"*. J. Comput. Appl. Math. 257 (2014) 195211.

[29] E. Muller, X. Guo, R. Scheichl, S. Shi, *"Matrix-free GPU implementation of a preconditioned conjugate gradient solver for anisotropic elliptic PDEs"*. Comput. Vis.Sci. 16 (2013) 4158.

[30] R. Helfenstein, J. Koko, *"Parallel preconditioned conjugate gradient algorithm on GPU"*. J. Comput. Appl. Math. 236 (2012) 35843590.

[31] M.M. Dehnavi, D.M. Fernndez, D. Giannacopoulos, *"Enhancing the performance of conjugate gradient solvers on graphic processing units"*. IEEE Trans.Magn. 47 (2011) 11621165.

[32] R. Li, Y. Saad, *"GPU-accelerated preconditioned iterative linear solvers"*. J. Supercomput. 63 (2013) 443466.

[33] G. Sharma, A. Agarwala, B. Bhattacharya, *"A fast parallel Gauss Jordan algorithm for matrix inversion using CUDA"*. Comput. Struct. 128 (2013) 3137.

[34] M. Ament, G. Knittel, D. Weiskopf, W. Strasser, *"A parallel preconditioned conjugate gradient solver for the Poisson problem on a multi-GPU platform"*, in: Proceedings of IEEE Conference on Parallel, Distributed and Network-Based Processing, Pisa, Italy, 2010, pp. 583592.

[35] K. Suresh, *"Efficient generation of large-scale pareto-optimal topologies"*. Struct. Multidiscip. Optim. 47 (2013) 4961.

[36] J. Martnez-Frutos, P.J. Martnez-Castejn, D. Herrero-Prez, *"Fine-grained GPU implementation of assembly-free iterative solver for finite element problems"*. Comput. Struct. 157 (2015) 918.

[37] Y. Cai, G. Li, H. Wang, *"A parallel node-based solution scheme for implicit Finite Element Method using GPU"*. Procedia Eng. 61 (2013) 318324

[38] V.R. Voller, C.R. Swaminathan, B.G. Thomas, *"Fixed grid techniques for phase change problems: a review"*. Int. J. Numer. Methods Eng. 30 (1990) 875898

[39] M.J. Garca-Ruz, G.P. Steven, *"Fixed grid finite elements in elasticity problems"*. Eng. Comput. 16 (1999) 145164.

[40] M.J. Garcia, M.A. Henao, O.E. Ruiz, *"Fixed Grid Finite Element Analysis for 3D linear elastic structures"*. Int. J. Comput. Methods 2 (2005) 569586.

[41] F.S. Maan, O.M. Querin, D.C. Barton, *"Extension of the fixed grid Finite Element Method to eigen value problems"*. Adv. Eng. Softw. 38 (2007) 607617.

[42] O.C. Zienkiewicz, R.L. Taylor, J.Z. Zhu, *"The Finite Element Method: Its Basis and Fundamentals"*. Elsevier Butterworth Heinemann, Oxford, 2013.

[43] J. Shewchuk, *"An Introduction to the Conjugate Gradient Method Without the Agonizing Pain"*. Technical Report, Carnegie Mellon University, Pittsburgh, PA,USA, 1994.

[44] M.J. Dayd, J.Y. L'Excellent, N.I.M. Gould, *"Element-by-element preconditioners for large partilly separable optimization problems"*. SIAM J. Sci. Comput. 18 (1997) 17671787.

[45] S. Bellavia, J. Gondzio, B. Morini, *"A matrix-free preconditioner for sparse symmetric positive definite systems and least-squares problems"* SIAM J. Sci. Comput. 35 (2013) 192211.

[46] Melenk, J.M., Gerdes, K., Schwab, C. *"Fully Discrete hp-Finite Elements: Fast Quadrature"*. Computer Methods in Applied Mechanics and Engineering, 1999/10/23.

[47] Cantwell, C. D., S. J. Sherwin, R. M. Kirby, and P. H. J. Kelly. *"From h to p efficiently: Strategy selection for operator evaluation on hexahedral and tetrahedral elements"*. Computers & Fluids vol. 43(1, SI), pp. 2328, 2011.

[48] CUDA C++ Programming, GuideB.12.URL: *"https://docs.nvidia.com/cuda/cuda-c-programming-guide /index.html#atomic-functions"* 2007-2019 NVIDIA Corporation.

[49] FreeFem++ - Universit Pierre et Marie Curie and Laboratoire Jacques-Louis Lions. URL: *"https://freefem.org/"* 1992, France

[50] Hestenes, Magnus R. Stiefel, Eduard (December 1952). *"Methods of Conjugate Gradients for Solving Linear Systems"*. Journal of Research of the National Bureau of Standards. 49 (6): 409. doi:10.6028/jres.049.044.

[51] Straeter, T. A. *"On the Extension of the Davidon-Broyden Class of Rank One, Quasi-Newton Minimization Methods to an Infinite Dimensional Hilbert Space with Applications to Optimal Control Problems"*. NASA Technical Reports Server. NASA. hdl:2060/19710026200.

[52] Speiser, Ambros *"Konrad Zuse and the ERMETH: A worldwide comparison of architectures"* (Konrad Zuse und die ERMETH: Ein weltweiter Architektur-Vergleich"),in: Hans Dieter Hellige (ed.): Geschichten der Informatik. Visionen, Paradigmen, Leitmotive. Berlin, Springer 2004, ISBN 3-540-00217-0. p. 185.

[53] Dominik Michels, Stefan Hartmannt *"Sparse-Matrix-CG-Solver in CUDA"*Institute of Computer Science II Rheinische Friedrich-Wilhelms-Universitat Bonn, Germany.

[54] Message Passing Interface *"URL: https://www.mcs.anl.gov/research/projects/mpi/"*

[55] Lei Chai, *"HIGH PERFORMANCE AND SCALABLE MPI INTRA-NODE COMMUNICATION MIDDLEWARE FOR MULTI-CORE CLUSTERS"* Doctor of Philosophy in the Graduate School of The Ohio State University, School of Computer Science & Engineering, Ohio, 2009.

[56] Erik R. Denlinger, *"Development and Numerical Verification of a Dynamic Adaptive Mesh Coarsening Strategy for Simulating Laser Power Bed Fusion Processes"* in Thermo-Mechanical Modeling of Additive Manufacturing, 2018.

[57] Wolfgang Bangerth, Timo Heister, Guido Kanschat, Matthias Maier et al. *"deal.II an open source finite element library"*, URL:https://www.dealii.org/, Heidelberg University, Germany 2000.

[58] NVIDIA TITAN V GPU *"URL: https://www.nvidia.com/en-us/titan/titan-v/"*, 2019.

[59] OpenACC *"URL: https://www.openacc.org/"*, 2012.

[60] OpenMP *"URL: https://www.openmp.org/"*, 1997.

[61] Kuusela, Mikael and Raiko, Tapani and Honkela, Antti and Karhunen, Juha, *"A gradient-based algorithm competitive with variational Bayesian EM for mixture of Gaussians"*. Proceedings of the International Joint Conference on Neural Networks, 1688-1695, 2009.

[62] Moore's Law *"URL: http://www.mooreslaw.org/"*, 1970.

[63] Moore's Law and Computer Processing Power *"URL: https://datascience.berkeley.edu/moores-law-processing-power/"*, Berkley School of Information, 2014.

[64] Breakdown of Dennard scaling around 2006 *"URL: https://en.wikipedia.org/wiki/Dennard_scaling#Breakdown_of_Dennard_scaling_around_2006"*, 2016.

[65] Michael M. Adams *"An Introduction to the C++ Programming Language"*, Department of Electrical and Computer Engineering, University of Victoria, 484-500, Canada, 2015.

[66] Article Reading 23: Locks and Synchronization in multi-processor programming, *"URL: web.mit.edu /6.005/www/fa15/classes/23-locks/"*, MIT, 2015.

[67] Hans-J. Boehm, *"Transactional Memory Should Be an Implementation Technique, Not a Programming Interface"*, HPC Laboratories - URL: https://www.usenix.org/legacy/event/hotpar09/tech/full_papers/boehm/boehm_html/index.html, 2010.

[68] Java - Thread Deadlock, Multithreading Guidelines, *"URL: https://www.tutorialspoint.com/java/ java_thread_deadlock.htm"*,Java Tutorials.

[69] Yifan Sun, Nicolas Bohm Agostini, Shi Dong, and David Kaeli, *"Summarizing CPU and GPU Design Trends with Product Data"*, Northeastern University, November 2019, USA.

[70] NVIDIA Titan V, *"URL: https://www.nvidia.com/en-us/titan/titan-v/"*.

[71] *"URL: https://forums.overclockers.co.uk/threads/titan-v-announced-2700-15tf-12nm-volta.18803276/page-8"*.

[72] CUDA Toolkit Documentation, *"URL: https://docs.nvidia.com/cuda/cublas/index.html"*, NIDIA, 2019.

[73] Alex Hutcheson and Vincent Natoli, *"Memory Bound vs. Compute Bound: A Quantitative Study of Cache and Memory Bandwidth in High Performance Applications"*. Stone Ridge Technology, 2011.